P A S C A L / M T

Release 2.0

User's Guide

Compiler, Debugger and Manual
Written by

Michael G. Lehman

Table of Contents

1.0 Introduction

Pascal/MT is a product for 8080/8085 and Z80 microcomputers. The Pascal/MT package consists of a compiler for a subset of the Standard Pascal language and a SYMBOLIC DEBUGGER. It will run in a 32K CP/M system (CP/M (tm) Digital Research) and under CP/M derivatives (CDOS,MCOS, etc.).

This manual is divided into two major sections. Sections 1,2,3,4,5,7 and 8 are designed for the experienced programmer. Section 6 is designed for the programmer who is familiar with some version of BASIC and would like to learn Pascal.

The remainder of section 1 describes the features of Pascal/MT and the requirements of the compile-time and run-time systems necessary to utilize Pascal/MT

1.1 Features

Pascal/MT is a subset of the Standard Pascal language described by Jensen and Wirth in the "Pascal User Manual and Report (2nd. ed.)". It is designed to provide a system which will run under CP/M and provide an efficient development cycle as well as efficiently executing object programs. To achieve this end Pascal/MT has extensions to the Standard Pascal language as well as some omissions.

Features of Pascal/MT

*** Compiler executes in 32K byte system
*** Run-time programs as small as 1.5K bytes
*** CP/M File interface at the block level
*** INPUT/OUTPUT pre-defined arrays for directly manipulating I/O ports from Pascal/MT without requiring assembly language routines
*** Logical manipulation features to allow bit manipulation
*** Assembly language interface to subroutines assembled under CP/M
*** Object Programs execute TEN TIMES FASTER than P-code systems
*** Package includes real-time SYMBOLIC DEBUGGER

Restrictions on the Pascal/MT language
(with respect to Standard)

- Ordinal and Record types not supported
- Reals not supported (coming in Release 3.0)
- GOTO not implemented
- Variables are allocated statically (therefore recursive calls do not mean new copy of variables allocated dynamically)
- Sets not supported

1.2 Compile-time system requirements

Pascal/MT will compile in a CP/M system as small as 32K. While it may be possible to execute the compiler in a smaller system the symbol table space will be very limited in a system under 32K. The operating system must be CP/M 1.3 or equivalent. This includes Cromemco CDOS and Digital Group MCOS. The product will be supplied on appropriate media for the system in question.

Pascal/MT is available on the following hardware:

*** All standard 8" CP/M systems
(IBM 3740 single density soft sectored)

*** Cromemco 8" and 5" CDOS

*** IMSAI 8" IMDOS

*** TRS-80 (with CP/M 1.4)

*** Industrial Micro Systems 5" CP/M

1.3 Run-time system requirements

Programs compiled by the Pascal/MT compile require at minimum 1.5K bytes of program storage area. Since the run-time package and the object code produced by the compiler is ROMable the ammount of RAM space is dictated only by the Pascal/MT program's variables. The Pascal/MT SYMBOLIC DEBUGGER requires 2.5K bytes of RAM storage in which to execute. The debugger will execute only in RAM. Programs written in Pascal/MT can execute in any CP/M system.

1.4 Contents of distribution disk

The Pascal/MT distribution disk contains the following files:

COMPILE.COM	- Pass 1 of compiler
P2	- Pass 2 of compiler
P1ERRORS.TXT	- Pass 1 error messages
P2ERRORS.TXT	- Pass 2 error messages
PASCAL.RTP	- Run-time package (including debugger)
FILEIO.SRC	- Sample routines for CP/M file Input/Output
LISTIT.SRC	- Sample program using FILEIO.SRC
DUMP.SRC	- Sample program
DUMPSUBS.SRC	- Include file for DUMP

2.Ø Compiler operation

The Pascal/MT compiler consists of two 8Ø8Ø object code files on the distribution disk: COMPILE.COM and P2. The input files to the Pascal/MT compiler must have the extension '.SRC' indicating that it is a source program file. The source file may be produced using a variety of editors (e.g. ED, WORD-MASTER, WORD-STAR, NED, etc.) and may include spaces, tabs, carriage returns, etc. Note: There must be a carriage-return/line-feed sequence at the end of each input line and an input line may not be longer than 8Ø characters.

2.1 Invocation

The Pascal/MT compiler is invoked using the following command:

```
COMPILE username
```

Where "username" is the name of the file with the extension of .SRC containing the Pascal/MT source statements to be compiled. There are no option switches used on the command line and data after the "username" is ignored.

The compiler will be loaded and display an identification message containing the release number and copyright notice. Following this the message:

```
Include debugger as part of final program?
```

will be displayed on the user's terminal. If the program is known to be good the user may answer N (or n) to indicate that the debugger and debugger information need not be included in the final .COM file. If the program is still being tested the user must answer Y (or y) to include the debugger and debugger information in the final .COM file. Including the debugger in the user's program will increase the size of the executable program by about 2.5K bytes.

Following this dialog the message:

```
Generate .PRN file?
```

will be displayed on the user's terminal. If the user types Y (or y) a .PRN file will be generated containing a number listing of the user's .SRC file (including all include files). This is useful when using line numbers in the debugger.

2.2 Compilation flow

The Pascal/MT compiler is a two-pass compiler. The first pass scans the syntax of the input program and converts the program to tokens suitable for use by the second pass. In this way the second pass need not worry about the correctness of the program while generating object code. The Pascal/MT compiler has a nominal speed of 6ØØ lines/minute. Very short programs will be much slower because of file manipulation overhead in CP/M. As the user's program becomes larger the percentage of the compilation time required for this overhead becomes smaller therefore the lines/minute throughput will increase.

When the user invokes the compiler using the COMPILE command this will load in the first pass of the compiler. The first pass will read the user's .SRC file and produce TEMP.PØ1. This will contain the tokenized program.

After the first pass is done it will load in the second pass from the P2 file. The second pass will read the TEMP.PØ1 file, and eventually produce a file called "username".COM (where username was supplied in the COMPILE command line input). Along the way the second pass will generate the object code into a file called TEMP.PØ2 and the symbol table into a file called SYMBOLS.%%%.

All of the temporary files are deleted at the end of a compilation.

In systems utilizing floppy disks for storage the compiler is practically bound by the speed of the disk during the first pass. The second pass is typically bound by the speed of the processor.

3.0 Pascal/MT Language Specification

Pascal/MT input source programs consist of Pascal statements and comments. Comments are bounded by '(*' or '{' on the left and '*)' or '}' on the right (nesting of comments is not allowed).

There is a special comment form which is used to supply compile-time options to the compiler. The form of these special comments is (* or { followed by $ followed by a letter followed by information (either a file name or +/-).

The $I form is used to copy source from a file different than the file named in the COMPILE command. This is called the INCLUDE file mechanism. The form of this special comment is (*$I <filename>*) (or {$I <filename>}). The file must reside on the logged in disk and have an extension of .SRC. An example of this is shown in the DUMP.SRC program on the distribution disk.

The $L form is used to turn the listing on and off during a compile to allow selective listing of the source program. This option is only active if a .PRN file is being generated. Otherwise it is ignored. The form of the $L comment is (*$L-*) to turn listing off and (*$L+*) to turn listing back on again.

The $P form is used to generate a form-feed in the .PRN file which will cause most printers to page eject when printing the listing. The form of the $P comment is (*$P*).

The list below describes a number of implementation details of Pascal/MT:

* Integer range is -32768..+32767

* Booleans are stored as 16-bit values
* The word PACKED is allowable but ignored
  CHAR arrays are always packed and BOOLEAN arrays are always unpacked

* ORD(TRUE) is 0001
* ORD(FALSE) is 0000

* Hexadecimal integers are allowed by prefixing them with a dollar sign (e.g. $FDEE or $0F)

* Characters are represented in standard ASCII

* The field width option on the WRITE statement

is not implemented in Release 2.Ø (to be done in Release 3.Ø)

* The reserved word FILE is not implemented use TEXT

3.1 Statements

Section 3 of this document describes the Pascal/MT language in detail. It is designed for readers who are already familiar with the Pascal language. Readers who wish to get started on Pascal may wish to read section 6 (Introduction to Pascal/MT for BASIC programmers) first in order to get a flavor of the language Pascal.

Subsections 3.1.1 through 3.1.13 describe the standard statements in the Pascal/MT language. For readers already very familiar with Pascal primarily only sections 3.1.7 (External Procedures), 3.2 (File Input/Output) and 3.3 (Special Procedures) will be of interest.

A Pascal/MT program consists of a number of lines of input source code which are loosely called statements. Every program must begin with the word PROGRAM and end with a single decimal point '.' . Sections 3.1.1 through 3.1.13 describe (in prose) the syntax of the Pascal/MT language. Section 8.0 contains a terse syntax diagram description of the entire language.

3.1.1 PROGRAM

Every Pascal/MT program must begin with the word PROGRAM. The word PROGRAM must be followed by an identifer and a semicolon. Following the program header the user may optionally declare constants, types, procedures, and functions. Following any such declarations the user must use the word BEGIN, followed by a series of statements separated by semicolons, followed by the word END and finally a decimal point (dot) '.' . The syntax for the statements and declarations are described in the sections which follow.

3.1.2 CONST

Typically user programs contains constant values such as control characters, port addresses, memory addresses, etc. In assembly language one would define these using some form of EQUate statement. In Pascal/MT a similar facility is provided for users to define constants. Constants of integer, character and boolean may be defined. Character constants are only allowed to be one character long. Unlike assembly language there are no expressions allowed in Pascal/MT constant definitions.

Constant definition in Pascal/MT is accomplished by using the word CONST followed by any number of the following group of symbols: an identifier, an equal sign, a constant value, and a semicolon. Listed below are some examples:

```
CONST
        consoledataport = 1;
        clearscreen = $1A;
        gocmd = 'G';
        debugging = TRUE;
```

3.1.3 TYPE

Pascal/MT is a typed language. This means that each user variable must be given an explicit type (e.g. INTEGER, CHAR, BOOLEAN, TEXT) and the compiler will perform type checking to validate the programs' use of variables. Users must use arguments of the proper type when calling procedures. In order to be able to pass arrays to procedures and functions Pascal/MT has user defined array types. (Note: While Standard Pascal has many more complex types Pascal/MT has only user defined array types for parameter passing). User defined types consist of the word TYPE followed by any number of the following group of symbols: an identifier, an equal sign, the word ARRAY, a left bracket, a subscript range specification, a right bracket, the word OF and a simple type (INTEGER, CHAR or BOOLEAN). Listed below are some examples:

```
TYPE
        coord = array [1..15] of integer;
        buffer = array [0..127] of char;
        options = array [1..5] of boolean;
```

3.1.4 VAR

All variables in Pascal/MT must be explicitly declared in a VAR statement (or in a parameter list, see 3.1.5,3.1.6). Declaration will establish a memory address, data type, length, and scope for the variables. Multiple variables of the same type can be grouped together. Variable declarations consist of the word VAR followed by any number of the following group of symbols: a list of identifiers separated by commas, a colon, a data type (INTEGER,CHAR,BOOLEAN,usertype) or a declaration of an array of a data type. Listed below are some examples:

```
VAR

        i : integer;
        ch: char;
        flag : boolean;

        ar1 : array [1..15] of integer;

        ar2 : utype1;
        { previously declared in a TYPE statement }
```

3.1.5 Procedures

In Pascal/MT user subroutines are divided into two types: procedures and functions. Functions are subroutines which return a value and may be used in an expression (e.g. SIN, COS, etc.). Procedures are subroutines which are called explicitly in separate statements.

Procedures and functions follow basically the same structure. The name of the procedure/function is optionally followed by a parameter list. This parameter list consists of names and types for the parameters to be passed through when this procedure/function is called. A parameter may be denoted as call by name (meaning that modification of the parameter within the procedure/function causes modification of the real variable) by using the word VAR before naming the parameter.

Local constants, types, variables and procedures / functions may be declared within a procedure / function definition. These constants, types, variables and procedures / functions are accessable only to the procedure / function being defined. This process of symbol table management is termed block structuring. Its primary purpose is to isolate from the "outside world" the inner workings of procedures (e.g. a hash function in a symbol table entry routine may be defined internal to the entry routine and therefore be changed without any other code requiring changes).

Listed below are some examples of procedure declaration. For function declarations see section 3.1.6.

```
PROCEDURE x;

PROCEDURE getnextchar(VAR ch:char);

PROCEDURE writenextchar(ch:char);
```

3.1.6 Functions

As described in section 3.1.5, functions are subroutines which return a value and may be used in an expression. Typical functions include SIN, COS, ABS, etc. In Pascal/MT functions may be typed as INTEGER, CHAR or BOOLEAN. Note: Because variables are allocated statically use of a function name in an expression within the function may give different results than expected. Listed below are some examples:

```
FUNCTION bittest(inp,mask:integer): boolean;

FUNCTION logicaland(a,b:integer): integer;

FUNCTION motorready(mot:integer): boolean;
```

3.1.7 External Procedures

Because there are some time dependent routines which may not be realizable in Pascal/MT and users may have already existing software written in assembly language, Pascal/MT has a facility for calling externally assembled procedures. Note that only procedures may be used in this way and not functions. In order to use this facility the user must assign an absolute address for the assembly language routine. In addition only three parameters (max) are allowed to EXTERNAL procedures. These are passed in the BC(#1), DE(#2), and HL(#3) registers. When the user's routine is entered there is a return address on top of the stack.

Listed below are some examples:

```
PROCEDURE EXTERNAL[5] BDOS(func:integer;addr:integer);

PROCEDURE EXTERNAL[$FCØØ] MOVEMOTOR(mot,posn:integer);
```

3.1.8 Assignment Statements

Assignment statements in Pascal/MT are exactly compatible with Standard Pascal but there are some extensions which make bit manipulation, direct memory accessing, and Input/Output more convienent.

There are operators &, ! and ~ (optionally \) which allow the logical operations on 16-bit integers. These operations are and (&), or (!) and not (~ {or \}). Not is treated similar to a leading minus sign but instead of a two's complement negate it generates a one's complement. and (&) and or (!) have similar precedence to the equivalent boolean operators.

Integers in Pascal/MT may be used as pointers to absolute memory locations. This will work for all data types except arrays. (For arrays see MOVE special procedure in section 3.3). This will allow the user to directly manipulate absolute memory locations by means of following an integer variable name with an up-arrow (e.g. ptr^). The value of the integer variable then becomes the address of the variable actually being accessed. This facility also relaxes type checking on the left-hand-side of assignment statements so that p^ := 'a' is quite legal.

In addition to bit manipulation and pointers there is one other feature to Pascal/MT assignment statements / expressions which is the ability to directly manipulate I/O ports in Pascal. This has been implemented as "magic" INPUT and OUTPUT arrays with a default declaration of Ø..255. The subscript used with these arrays must be an explicit or user defined (named) constant. These arrays are untyped and therefore may be used with integers, characters or booleans. INPUT may be used only in expressions and OUTPUT may be used only on the left-hand-side of an assignment statement.

While these extensions are somewhat outside the scope of normal usage of Pascal every effort has been made to include all the features necessary to write useful programs in the Pascal/MT language in order to eliminate 9Ø% of the need to drop into assembly language routines.

In all other respects assignment statements are exactly the same as in Standard Pascal.

3.1.9 IF...THEN...ELSE

This statement form is exactly the same as Standard Pascal.

3.1.10 CASE...OF

This statement form is exactly the same as Standard Pascal.

3.1.11 WHILE...DO

This statement form is exactly the same as Standard Pascal.

3.1.12 REPEAT...UNTIL

This statement form is exactly the same as Standard Pascal.

3.1.13 FOR...(TO/DOWNTO)...DO

This statement form is exactly the same as Standard Pascal.

3.2 File Input/Output

Standard Pascal READ/READLN/WRITE/WRITELN statements are implemented for the CP/M console device. For disk I/O there are six special extensions to the Pascal language implemented to handle CP/M files:

```
OPEN(filename,title,result);
CLOSE(filename,result);
CREATE(filename,title,result);
DELETE(filename);

BLOCKREAD (filename,buffer,result {,relativeblock});
BLOCKWRITE(filename,buffer,result {,relativeblock});
```

```
filename : a variable of type TEXT
title    : ARRAY [Ø..11] OF CHAR; (* CP/M file name *)
result   : integer to contain returned value from BDOS
buffer   : ARRAY [Ø..127] OF CHAR; (* one sector *)
relativeblock : optional integer; default is sequential
```

A variable of type TEXT is in reality an array Ø..32 of CHAR. Subscript 12 is the extent number and subscript 32 is the Next Record (NR) field. See the CP/M Interface Guide for a complete description of the FCB format.

3.3 Special Procedures

Because there are some occasions where the user wishes to do a function repeatedly which is not defined tne Pascal language, Pascal/MT has included the following special procedures in the Pascal/MT system:

1.

MOVE(source,destination,length-in-bytes);

source can be array name, or integer used as pointer

destination (same as source)

(source or destination arrays may be subscripted)

length-in-bytes is an integer expression

2.

EXIT

This will exit the current procedure / function or the main program. It is the equivalent of the RETURN statement in FORTRAN or BASIC.

4.0 Interactive Symbolic Debugger operation

One of the key features of the Pascal/MT package is the inclusion of an interactive symbolic debugger. Experience has shown that at least 50% of the compilations done in the course of developing a Pascal program are to include debugging WRITELN statements in order to gain visibility into the operation of the program under development. The Pascal/MT Interactive Symbolic Debugger solves this time consuming problem by providing both program flow and variable data visibility at run-time repeatedly without requiring re-compilation.

As noted in section 2.1 the compiler will ask the user about the inclusion of the debugger in the final .COM file. If the user requested the debugger then all variables and program flow information will be available at run-time.

The facilities available to the user when using the debugger fall into two catagories: program flow control and variable display. Sections 4.1 through 4.1.6 describe the program flow control commands available and section 4.2 describes the variable display command.

Whenever a program is being executed with the debugger the user may return to the debugger (while the program is executing) by typing two rubouts.

If the user wishes to see the commands during the execution of the debugger the user need only to type ? followed by carriage return and a summary of the commands will be output to the console.

4.1 Program flow commands

The program flow commands provided in the symbolic debugger allow the user to debug the Pascal/MT program at the Pascal source statement level. Included are go/continue (with optional breakpoint), trace (execute an arbitrary number of lines), set/clear/display permanent breakpoint and a mode which will display the name of each procedure/function on the console as the procedure/function is entered. These commands are described in detail in sections 4.1.1 through 4.1.6.

4.1.1 G - Go (with optional breakpoint)

The G (Go) command allows the user to resume execution of the program under test from where it stopped. Optionally the user may set a breakpoint at a specific line number or entry to a procedure / function. The line numbers correspond to the lines in the source program. When the breakpoint is reached control will return to the debugger so that the user can issue further commands.

The syntax of the G command is:

G{,<linenumber>}

or

G{,<proc/func name>}

4.1.2 T - Trace

The T (Trace) command allows the user to execute one or more lines of the program under test while maintaining complete control of the program in the debug mode. The T command if issued without any arguments will trace one line. If an integer is placed after the T command that many lines will be executed before control will return to the operator. As noted above if the user wishes to regain control before the number of lines requested have been executed the user may type two rubouts.

The syntax of the T command is:

T{<integer>}

4.1.3 E - Procedure/Function Display Toggle

One of the most common problems found when debugging a program in a higher level language is lack of visibility with regards to the flow of control within the program. Users have resorted to placing output statements at the begining of each subroutine which will allow the programmer to follow the program execution. Pascal/MT solves this problem by having this flow display built into the compiler/debugger combination. When a user program is compiled and the debugger is requested the compiler will generate a call to a debugger routine at the begining of each procedure to optionally write out a message identifying the procedure/function being entered. The E (Entry display control) command allows the programmer to dynamically enable and disable this feature.

If the E command is entered as just 'E', the display will be enabled. If it is preceeded by a minus sign '-E' the display will be disabled.

When the Entry display mode is enabled the message:

```
Entering : xxxxxxxx
```

will be displayed on the console device each time a procedure/function is entered.

On a high-speed CRT terminal this typically happens very quickly. Section 4.1.4 describes the Slow exec mode control command which causes the debugger to pause (for a variable length of time) following each display of the above message so that the user will have time to read and react. Remember that typing rubout twice will bring control back to the debugger if in Go or Trace mode.

4.1.4 S - Set/Clear Slow execution mode

The S (Slow exec mode control) command allows the user to set a time delay which will be used after the 'Entering Procedure' message described in section 4.1.3. This time delay is used so that the user will have time to read the message and react. There are two modes used with the S command. If the command is preceeded by a minus sign '-S' then this will cause the user program to operate at full speed. If the command is issued without a minus sign 'S' then the debugger will display the following message:

```
F(ast), M(edium) or S(low)?
```

The user should respond with F,M or S as desired. F(ast) mode causes a delay of approximately one second after the 'Entering Procedure' message. M(edium) mode causes a delay of two seconds and S(low) mode causes a delay of four seconds.

Note that these times are for a 2MHz 8080 and will be half as long on a 4MHz Z80.

4.1.5 P - Set/Clear permanent breakpoint

There are times when debugging programs that the programmer wishes to stop at the same place repeatedly. While this can be accomplished using the G command typing the line number at which to stop quickly becomes tiresome. In order to facilitate this mode of operation the Pascal/MT debugger has a "permanent breakpoint" mode. In this mode the user may set a "permanent" breakpoint at a specific line number or procedure / function entry. Each time the program under test executes that line control will return to the debugger. The user may then set/clear toggles, display variables, etc. The next G command will cause execution to resume with the line after the breakpoint. When this line is again executed control will return to the debugger. This is very handy when attempting to debug a problem in the 15th time through a loop.

This "permanent" breakpoint can also be cleared by typing a minus sign before the command '-P'. The syntax of the P command is:

```
-P                  {clears permanent breakpoint}
P<linenumber>       {sets the permanent breakpoint}
or
P<proc/func name>   {sets the permanent breakpoint}
```

4.1.6 B - Display permanent breakpoint

If the user has set the permanent breakpoint (see section 4.1.5) the user may also wish to display what line the permanent breakpoint is set for. This is the function of the B command. This command will cause the following line to be output on the console:

```
permanent breakpoint is: nnnn
```

(if the breakpoint is symbolic the message below will also appear)

```
Symbolic stop at: xxxxxxxx
```

4.2 Variable display command - D

One of the most powerful features of the Pascal/MT debugger is the ability to display variables in the program under test at run-time by NAME. The variables are displayed in the format in which the user would expect (i.e. decimal for integers, character form for characters, TRUE/FALSE for booleans).

Because Pascal/MT allocates all variables statically the user is able to display any variable at any time even if the procedure/function containing a variable is not currently active.

Variable names used with the debugger take one of two forms. The name of the variable is all that is necessary to display global variables and constants. If a variable is local to a procedure/function (i.e. in a parameter list or defined in a CONST/VAR statement inside a procedure/function) then the user must qualify the name of the varaible with the name of the procedure. This is done by naming the procedure followed by a colon followed by the variable name.

Some examples of using the D command are shown below:

```
D I                 {display global variable I}
D A:I               {display variable i in procedure/function A}
D F1                {display value of function F1}
D P^                {display integer at location contained in P}
```

5.Ø Run-Time environment

Pascal/MT programs require a run-time package in order to operate. This run-time package is contained in the file PASCAL.RTP on the distribution disk. This file contains all of the routines described in section 5.1 as well as the object code for the Interactive Symbolic Debugger.

5.1 Standard Routines

The following routines expect two operands on the stack underneath the return address. Where the order of the operands is important they will appear as B followed by A as in A DIV B or A - B:

Address	Name	
0103	DIV	Two's complement divide
0106	MUL	Two's complement multiply
0109	MOD	Two's complement divide (return quotient)
012D	INTEQ	Integer equal compare return with TRUE/FALSE on stack
0130	INTNE	Integer not equal compare
0133	INTGT	Integer greater than compare
0136	INTLT	Integer less than compare
0139	INTGE	Integer greater than or equal compare
013C	INTLE	Integer less than or equal compare

The remainder of the routines take various differing parameters: TOS refers to the first word under the return address

010C	WINT	Write integer, TOS = integer
010F	RINT	Read integer, HL = addr of result area
0112	WCHR	Write string, TOS = addr of string, HL = length
0115	CHRW	Write single char, TOS = char in low byte
0118	RCHR	Read a single char, HL = addr of result area
0127	CRLF	Write CR then LF on console
012A	CRWAIT	Wait for user to type CR
013F	CASEJMP	Scan case table for match to argument on TOS DE = addr of case table
0142	FORASSIST	Calculate (TOS)-(TOS+2) to HL (used to calc number of times to go in a for loop)
0145	BLKMOVE	Move a number of bytes, BC = len, DE = ^src, HL = ^dst
014E	CHREQ	Compare char string for = HL = ^str1, DE = ^str2, BC = length (push TRUE/FALSE)
0151	CHRNE	Compare char string for <>
0154	CHRGT	Compare char string for >
0157	CHRLT	Compare char string for <
015A	CHRGE	Compare char string for >=
015D	CHRLE	Compare char string for <=

The following routines interface to CP/M:

Ø118	OPEN	HL = addr of fcb,DE=addr of title, BC=addr of result
Ø11E	CLOSE	HL = addr of fcb,BC = addr of result
Ø148	DELETE	HL = addr of fcb
Ø14B	CREATE	HL = addr of fcb,DE=addr of title, BC=addr of result
Ø121	FREAD	same as FWRITE
Ø124	FWRITE	HL = addr of fcb,DE=addr of buffer, BC=addr of result

The following routines are used by the compiler to gain access to the debugger:

Ø160	CHKBPT	Check for breakpoint. Following a call to CHKBPT is the line number to check for
Ø163	PROCENT	Optionally display proc/func name. Following the call to PROCENT are 8 chars to display

6.Ø Introduction to Pascal/MT for BASIC programmers

While everyone continues to rave about Pascal as a programming language the microcomputer community, in general, is still programming in one dialect of BASIC or another. This section of the Pascal/MT User's Guide is intended to bring BASIC programmers up to speed on the fundamentals of Pascal.

This is not intended to be a complete coverage of the entire language but simply a survival kit which can be used to get started. As all programmers learn quickly the only way to learn a new language is to get one's feet wet.

6.1 General

Pascal/MT has the capability of performing any task which can be performed in BASIC. The single, most outstanding, difference between BASIC and Pascal is that all variables must be declared in Pascal. While this may seem troublesome at first it quickly becomes obvious that one no longer has to ask questions like "What does Q$ or V1 mean and what is it used for?". Variable names is Pascal are significant to 8 characters but may be as long as the user wishes. Examples of Pascal variable names are: CONSOLESTATUSPORT, INPUTBUFFER, SYMBOLTABLEARRAY, etc. This feature significantly increases the readability of a program. While this may seem like a problem at first, everyone knows of some program written six months ago which is now totally indecipherable as if it was written in Etruscan.

Another major difference between Pascal and BASIC is that there are no line numbers to be used for GOTOs or GOSUBs. All forward and backward branching is done with IF..THEN..ELSE, REPEAT..UNTIL, WHILE..DO, FOR and procedure calling statements. This is what is currently being called STRUCTURED PROGRAMMING because it allows the reader of a program to determine some of the structure of the program by simply looking at the listing as opposed to trying to follow a rat's nest of GOTO and GOSUB statements.

One of the great advantages of Pascal over BASIC is that procedures can be named and parameters can be passed. This allows the user to really concentrate on the job in progress rather than how to call a subroutine with a different argument every time.

The remainder of section 6 is devoted to examples of common sequences in BASIC and their equivalent in Pascal/MT. This is organized around the BASIC statement types. Note that comments in Pascal are surrounded by (* on the left and *) on the right and may cross line boundaries.

6.2 Equivalent statements

6.2.1 LET

The most basic of all BASIC statements is the LET statement. This allows the user to perform calculations and assign the results to a variable. As we noted in section 6.Ø all variables in Pascal must be declared before use. The syntax of expressions on the right-hand-side of an assignment statement is the same in Pascal as it is in BASIC.

Some examples are listed below:

BASIC	PASCAL
(1) LET X = 3	X := 3;
(2) LET P$ = "C"	P := 'C';
(3) LET Y = (I+2)*1Ø	Y := (I+2)*1Ø;
(4) LET X(3) = Y(J+1)	X[3] := Y[J+1]

One difference can be noted in example 4 in which Pascal uses square brackets to denote array subscripts.

One feature that Pascal has that is not present in BASIC is called type-checking. This means that the compiler will prevent the programmer from cheating either on purpose or accidentally. However, there are "pseudo" functions which the programmer can use to tell the compiler that the programmer is aware that cheating is taking place. These "pseudo" functions allow the assignment of a character to an integer, an integer to a character, a boolean to an integer, and so forth. A complete description of how these work can be found in Jensen and Wirth. Some examples are:

```
A := ORD('R');
(* assign the binary value of the char R to A *)

C := CHR(3);
(* assign the binary value 3 to the char C *)

FLAG := ODD(A);
(* assign the boolean value of A to FLAG *)
```

6.2.2 GOTO

There is no equivalent in Pascal/MT of the BASIC GOTO statement. Forward and backward branching is done using control constructs such as IF/THEN/ELSE, REPEAT/UNTIL, etc.

6.2.3 IF

The Pascal form of the IF statement is much more sophisticated than the IF statement in most forms of BASIC. The Pascal syntax is:

```
IF <boolean_expression> THEN
  statement
```

and optionally

```
ELSE
  statement
```

If the programmer wishes to place multiple statements after the THEN or the ELSE the statements must be surrounded by BEGIN and END and separated by semicolons. The term <boolean_expression> is traditionally the form expression relop expression (e.g. A = B) but can also include logical operations on boolean variables (e.g. DEBUGGING AND PASS2 {where DEBUGGING and PASS2 are boolean variables}). Some examples of Pascal IF statements are shown below:

```
IF A = B THEN
  WRITELN('A EQUALS B')
ELSE
  WRITELN('A IS NOT EQUAL TO B');


IF DEBUGGING AND PASS2 THEN
  ENTRYCOUNT := ENTRYCOUNT + 1;


IF (A=B) OR (C=D) THEN
  E := F;
```

6.2.4 GOSUB / RETURN

The BASIC GOSUB statement is not found directly in Pascal. Procedures are declared with the PROCEDURE statement and then called simply by naming them as a statement. This makes the program very easy to write as well as read. In addition functions can be declared in Pascal and be used in expressions. Functions are declared to return a specific type of value. Some examples of Procedure/Function declarations and usage are found in the sample program below. If the user wishes to leave the procedure / function early (i.e. before returning at the end) the user may use the EXIT special procedure. (see section 3.3 for details).

```
PROGRAM procfuncexample;

VAR
  CH : CHAR;


  PROCEDURE waitforcharacter;

  BEGIN
    WHILE INPUT[3] = Ø DO        (* wait for data available *)
  END;


  FUNCTION getcharacter : char;

  BEGIN
    getcharacter := INPUT[4]
  END;


BEGIN (* main program *)

  waitforcharacter;
  ch := getcharacter

END.
```

6.2.6 DIM

In Pascal, just like in BASIC, all arrays must be declared before using. Pascal does this along with all other declarations using the VAR statement. Like all other variables in Pascal arrays must have a type.

In order to pass arrays as parameters to procedures the TYPE statement is used. This allows the programmer to define an array type and give it a name. This type name is then used to declare the variable and is also used in the parameter list.

Some examples are shown below:

```
(* Example 1 *)

       VAR

          inputbuffer : array [1..15] of char;
          list : array [Ø..25] of integer;


(* Example 2 - passing an array as a parameter *)

       TYPE

              coordinatearray = array [Ø..511] of integer;

       VAR

              cameral : coordinatearray;
              camera2 : coordinatearray;

       PROCEDURE processarray(VAR x : coordinatearray);
       BEGIN
         .
         .
         .
       END;


       BEGIN (* main program *)

         processarray(cameral);
         processarray(camera2)

       END.
```

6.2.7 INPUT

The equivalent to the BASIC INPUT statement in Pascal is the READ statement. The READ statement may be used to input either integers or characters. Some examples:

```
READ(I,J,K);
READ(CH);
```

There is an optional form of the READ statement called READLN. After performing the desired input the program will wait for the user to type <return> before continuing. Note that if the last char typed during an integer input was a carriage return this is sufficient to end a READLN (i.e. two <return>s are not required).

6.2.8 PRINT

The equivalent to the BASIC PRINT statement in Pascal is the WRITE statement. The write statement may be used to output integer, character or character array variables. The WRITELN form of the WRITE statement should be used to advance to a new line.

Some examples:

```
WRITE('INPUT VALUE FOR A: ');

WRITELN('Program xxxxxx version 2.Ø');

WRITELN(A,B,C)
```

6.2.9 PEEK/POKE

In Pascal/MT there are features which are equivalent to PEEK and POKE found in some dialects of BASIC. In Pascal/MT one uses an integer variable as a "pointer" to the data desired. Following the integer variable name with an uparrow (^) will cause the data at the memory location contained in the integer to be used rather than the contents of the integer. This is called indirection. This feature in only valid in assignment statements for integers and characters. For character arrays see section 3.3 for a description of the MOVE special procedure.

Some examples:

```
VAR

  P,Q : integer;

BEGIN
        P := $4000;      (* hex 4000 *)
        P^ := 5;         (* put a word of 0005 at location 4000 *)

        Q := P^;         (* get word from loc 4000 to Q *)
```

6.2.1Ø ON...GOTO / ON...GOSUB

There is a statement in Pascal which is equivalent to both ON...GOTO and ON...GOSUB. This is the CASE statement. This allows the user to perform one of a number of statements based upon an expression. Unlike BASIC the CASE statement can have selectors which range over a wide set of values without having to use up line numbers. Some examples:

```
CASE ch OF

  'G' : gocmd;  { call a procedure }

  'S' : val := substitute(inp); { call a function }

  'M' : x := 'q';          { assignment statement }

  'Q' : BEGIN
          x := 'f';
          exit             { do multiple statements }
        END

END;                { end of the CASE statement }
```

7.0 Future Products

MetaTech is constantly developing new products to support more effective use of computer technology.

Future enhancements to Pascal/MT which are under development are:

Reals	Release 3.0	(November 1979)
Interrupt Procedures	Release 3.0	
MultiTasking	Release 3.0	
8086 support	Release 4.0	(January 1980)
Z8000 support	Release 4.1	(February 1980)

Additional structured programming tools and special purpose language construction tools are also in development. In particular a structured programming screen-oriented editor is scheduled for release in January of 1980.

8.Ø Syntactic description of Pascal/MT

The following notation is used:

Names in all caps (e.g. BEGIN) are reserved words.

Items separated by ! are treated as or (e.g. '+' ! '-')

Special characters are enclosed in quotes

Groups of symbols are enclosed in square brackets (e.g. [a ! b])

The symbol $ indicates "zero or more of" (e.g. $[';' statement])

Symbols in braces mean that they are optional (e.g. {ELSE statment})

Items following one another imply required order
(e.g. BEGIN statement $[';' statement] END)

The word 'ident' is used to mean a standard Pascal identifier

---

<program>

PROGRAM ident ';' block '.'

<block>

{constdecl} {typedecl} {vardecl} $[procdecl ! funcdecl] compound

---

<constdecl>

CONST $[ident '=' [charconst ! intconst] ';']

<typedecl>

TYPE $[ident = ARRAY '[' intconst '..' intconst ']' OF simpletype ';']

<vardecl>

VAR $[ident ':' arraydecl ! simpletype ';']

<arraydecl>

(note arrays of TEXT not allowed)

ARRAY '[' intconst '..' intconst ']' OF simpletype

<simpletype>

TEXT ! INTEGER ! CHAR ! BOOLEAN

<procdecl>

PROCEDURE {EXTERNAL '[' intconst ']'} ident {parmlist} ';' block

---

<funcdecl>

(note only INTEGER, CHAR and BOOLEAN types allowed)

FUNCTION ident {parmlist} ':' simpletype ';' block

<compound>

BEGIN statement $[';' statement] END

<statement>

compound ! ifstmt ! casestmt ! whilestmt ! rptstmt ! forstmt ! proccall ! assign

---

<ifstmt>

IF expr THEN statement {ELSE statement}

<casestmt>

CASE expr OF stmts END

```
------------------------------------------------------

<stmts>

case $[ ';' case]

------------------------------------------------------

<case>

const $[ ',' const]  ':'  statement

------------------------------------------------------

<whilestmt>

WHILE  expr  DO  statement

------------------------------------------------------

<rptstmt>

REPEAT  statement  $[ ';' statement]  UNTIL  expr

------------------------------------------------------

<forstmt>

FOR  ident  ':='  expr  [ TO  !  DOWNTO  ]  expr  DO  statement

------------------------------------------------------

<assign>

[ OUTPUT '[' intconst ']' !
  var                     !
  funcid                  ]  ':='  expr

------------------------------------------------------

<proccall>

procid { '(' expr $[ ';' expr] ')' }

------------------------------------------------------

<parmlist>

'('  parm  $  [';'  parm]  ')'

------------------------------------------------------

<parm>
```

```
{VAR}  ident  ':'  [  simpletype  !  ident  ]
```

```
<expr>
string  !  arithexpr  $[ relop arithexpr]
```

```
<relop>
'='  !  '<>'  !  '>'  !  '<'  !  '>='  !  '<='
```

```
<arithexpr>
['+' ! '-' ! '~']  term  $[  ['+' ! '-' ! OR ]  term  ]
```

```
<term>
factor $[ ['&' ! AND ! * ! DIV ! MOD]  factor ]
```

```
<factor>
intconst  !  boolconst  !  specialfunc  !  input  !
funccall  !  var  !  '('  expr  ')'
```

```
<intconst>
(preceeding number with '$' means number is base 16)
{'$'} digit  $[ digit ]
```

```
<digit>
('A' .. 'F' valid only if number preceeded by '$')
'Ø' ! '1' ! '2' ! '3' ! '4' ! '5' ! '6' ! '7' ! '8' ! '9'
! 'A' ! 'B' ! 'C' ! 'D' ! 'E' ! 'F'
```

```
<boolconst>
```

```
TRUE  !  FALSE
-------------------------------------------------
<specialfunc>
[ ORD  !  CHR  !  ODD  !  ABS ]  '('  expr  ')'
-------------------------------------------------
<input>
INPUT '[' intconst ']'
-------------------------------------------------
<funccall>
funcid {'(' expr $[ ',' expr ] ')'}
-------------------------------------------------
<var>
ident  { '['  expr  ']'  }
-------------------------------------------------
<charconst>
"'"  singlechar "'"
-------------------------------------------------
```